SMART CONTRACT AUDIT REPORT

for

dYdX Bridge

Prepared By: Xiaomi Huang

PeckShield

September 3, 2023

Document Properties

Client	dYdX
Title	Smart Contract Audit Report
Target	dYdX Bridge
Version	1.1
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	September 3, 2023	Xuxian Jiang	Post Release #1
1.0	August 31, 2023	Xuxian Jiang	Final Release
1.0-rc1	August 6, 2023	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4

1.1 About dYdX Bridge 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 6

2 Findings 9

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results 11

3.1 Adaptive Domain Separator in BridgeDydxToken 11

3.2 Revisited Inheritance for TreasuryBridge 13

3.3 Trust Issue of Admin Keys 14

4 Conclusion 16

References 17

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `dYdX Bridge`, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well-designed and engineered. This document outlines our audit results.

1.1 About dYdX Bridge

`dYdX Foundation`("Foundation") is an independent Swiss foundation created to support the `dYdX` ecosystem, including a leading decentralized protocol that supports perpetuals trading, towards community-led growth, development, and self-sustainability. The creation of the `dYdX Bridge` was commissioned by `Foundation`. The audited `dYdX Bridge` could facilitate a one-way transfer mechanism of the current `DYDX` token to a v4 chain (if deployed) run on open source software developed by `dYdX Trading Inc.`("Trading").

Table 1.1: Basic Information of dYdX Bridge

Item	Description
Name	dYdX
Website	https://dydx.exchange/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 3, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the audit scope only covers the following contracts: `contracts/governance/bridge/BridgedDydxToken.sol`, `contracts/governance/strategy/GovernanceStrategyV2.sol`, and `contracts/treasury/TreasuryBridge.sol`

- https://github.com/dydxfoundation/governance-contracts/pull/49 (7ca15b0)

And this is the commit ID after all fixes for the issues found in the audit have been checked in: Note `contracts/governance/bridge/BridgedDydxToken.sol` is renamed to `contracts/governance/bridge/WrappedEthereumDydxToken.sol` in commit `780808c`.

- https://github.com/dydxfoundation/governance-contracts/pull/49 (bc77ea6)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `dYdX Bridge`. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	0	
Low	3	■■■
Total	3	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 low-severity vulnerabilities.

Table 2.1: Key dYdX Bridge Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Adaptive Domain Separator in BridgeDydxToken	Business Logic	Confirmed
PVE-002	Low	Revisited Inheritance for TreasuryBridge	Coding Practices	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Adaptive Domain Separator in BridgeDydxToken

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `BridgeDydxToken`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `BridgeDydxToken` token contract strictly follows the widely-accepted `ERC20` specification. In the meantime, we notice the support of `EIP-2612` with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` is initialized once inside the `constructor()` function (lines 90-98).

```
78    constructor(
79        ERC20 tokenAddress
80      )
81        ERC20(NAME, SYMBOL)
82      {
83        uint256 chainId;
84
85        // solium-disable-next-line
86        assembly {
87          chainId := chainid()
88        }
89
90        DOMAIN_SEPARATOR = keccak256(
91          abi.encode(
92            EIP712_DOMAIN,
93            keccak256(bytes(NAME)),
94            keccak256(bytes(EIP712_VERSION)),
95            chainId,
96            address(this)
97          )
98        );
```

```

        DYDX_TOKEN = tokenAddress;
    }
```

Listing 3.1: `BridgeDydxToken::constructor()`

The `DOMAIN_SEPARATOR` is used in the `permit()` function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this `permit()` routine, we realize the current implementation needs to be improved by recalculating the value of `DOMAIN_SEPARATOR` inside the `permit()` function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed `DOMAIN_SEPARATOR`, a valid signature for one chain could be replayed on the other.

```
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        require(owner != address(0), 'INVALID_OWNER');
        require(block.timestamp <= deadline, 'INVALID_EXPIRATION');
        uint256 currentValidNonce = _nonces[owner];
        bytes32 digest = keccak256(
          abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
                currentValidNonce, deadline))
          )
        );

        require(owner == ecrecover(digest, v, r, s), 'INVALID_SIGNATURE');
        _nonces[owner] = currentValidNonce.add(1);
        _approve(owner, spender, value);
    }
```

Listing 3.2: `BridgeDydxToken::permit()`

Note the `DOMAIN_SEPARATOR` is also used in the `delegateByTypeBySig()/delegateBySig()` routines where there is a need to recalculate the `DOMAIN_SEPARATOR` value.

Recommendation Recalculate the value of `DOMAIN_SEPARATOR` before it is used in the `permit()/delegateByTypeBySig()/delegateBySig()` routines.

Status The issue has been confirmed.

3.2 Revisited Inheritance for TreasuryBridge

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Treasury/TreasuryBridge`
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

In the `dYdX` protocol, the `TreasuryBridge` contract inherits from the `Treasury` contract for the functionalities to approve/transfer `ERC20` tokens from the contract. While reviewing the inheritance implementation in the `TreasuryBridge` contract, we notice the syntax issues that should be corrected.

To elaborate, we show below the code snippet from the `TreasuryBridge` contract. The contract implements an `initialize()` routine for contract initialization and a `getRevision()` routine returning the current version number for the version check in the `initializer` modifier.

```
contract TreasuryBridge is Treasury
{
  ...
  function initialize()
    external
    initializer
  {
    TREASURY_VESTER.claim();
    TREASURY_VESTER.setRecipient(BURN_ADDRESS);
  }

  function getRevision() internal pure override returns (uint256) {
    return 2;
  }
  ...
}
```

Listing 3.3: `TreasuryBridge.sol`

While reviewing the following code snippet from the parent `Treasury` contract, we notice it also implements a copy of the `initialize()/getRevision()` routines. However, these two routines are not properly marked `virtual`. As a result, they are conflicted with those implemented in the child `TreasuryBridge` contract. Accordingly, the `TreasuryBridge::initialize()` routine should be properly marked `override`.

```
  contract Treasury is
  OwnableUpgradeable,
  VersionedInitializable
{
  using SafeERC20 for IERC20;
```

```
  uint256 public constant REVISION = 1;

  function initialize()
    external
    initializer
  {
    __Ownable_init();
  }
  ...
  function getRevision() internal pure override returns (uint256) {
    return REVISION;
  }
}
```

Listing 3.4: `Treasury.sol`

What's more, in the `TreasuryBridge::initialize()` routine, there is a need to call the `__Ownable_init()` routine to initialize the owner of the contract.

Recommendation Mark the `Treasury::initialize()/getRevision()` routines `virtual`, mark the `TreasuryBridge::initialize()` routine `override`, and call the `__Ownable_init()` in the `TreasuryBridge::initialize()` routine.

Status This issue has been fixed in the following commit: `780808c`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `dYdX` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the governance protocol-wide operations (e.g., transfer tokens from the treasury). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `Treasury` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the `owner` account is privileged to approve any recipient to transfer tokens from the treasury, and transfer tokens from the treasury to any recipient, etc.

```
function approve(
  IERC20 token,
  address recipient,
  uint256 amount
)
  external
  onlyOwner
{
  // SafeERC20 safeApprove() requires setting the allowance to zero first.
  token.safeApprove(recipient, 0);
  token.safeApprove(recipient, amount);
}

function transfer(
  IERC20 token,
  address recipient,
  uint256 amount
)
  external
  onlyOwner
{
  token.safeTransfer(recipient, amount);
}
```

Listing 3.5: The Privileged Operations in `Treasury`

We emphasize that the privilege assignment is necessary and consistent with the intended design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The audit presumes that this privileged account will be managed by a governance `DAO`. It should be noted that a compromised `owner` account would allow the attacker to mess up internal records and claim rewards for others, which directly undermines the assumption of the staking support.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been resolved and the audit presumes that the `owner` would be the `DAO` governance.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `dYdX Bridge`. The `dYdX Bridge` could facilitate a one-way transfer mechanism of the current `DYDX` token to a v4 chain (if deployed) run on open-source software which is developed by Trading. The current code base is well structured and neatly organized. Those identified issues were promptly confirmed and fixed.

Furthermore, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedback or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.